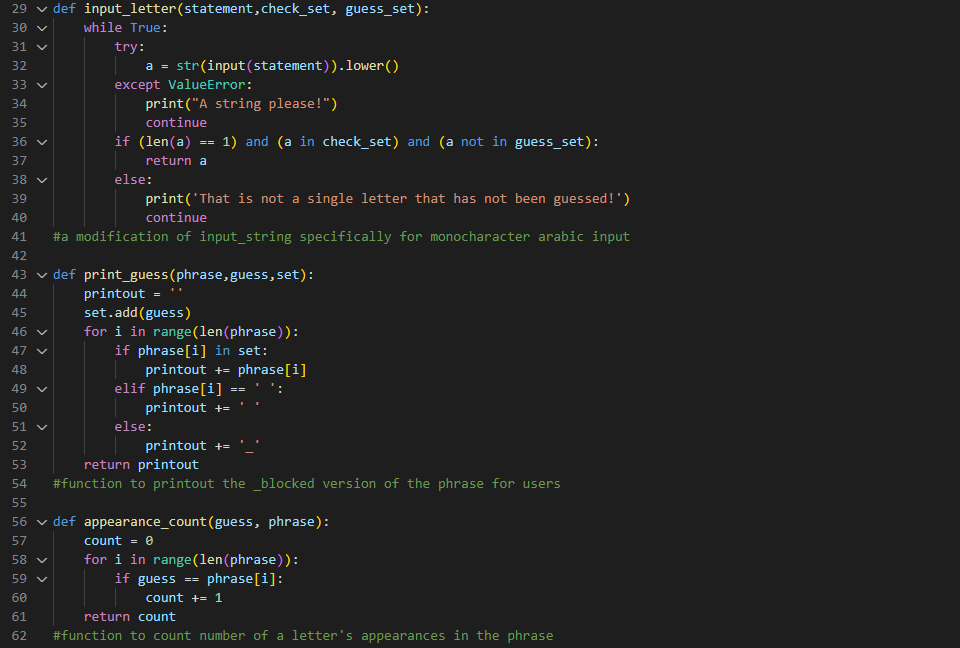
**The Driver:**



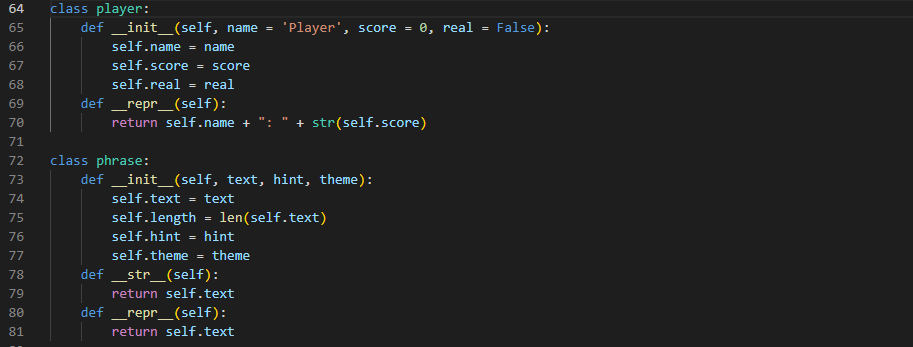
Here, we see intro data, as well as two functions borrowed from previous works, checking the evilness of user inputs of strings or integers without crashing the program. These programs are updated from previous versions, however, as this iteration has a (phrase) input to inform the user of what the input is being used for, without having to use an external print function before the function is run.



Built on a similar framework, we have the input\_letter function that is designed specifically for this game, as it checks to make sure that the user’s guess is not a letter that has already been guessed by a previous user. I chose to design the game with this feature (instead of penalizing with 0 points and moving on) so as to make the game move faster. Even when displaying all the previously guessed letters, they can be an awful lot to visually sift through, and I didn’t want to force players to scan over and over again to make *sure* that their guess had not already been used.

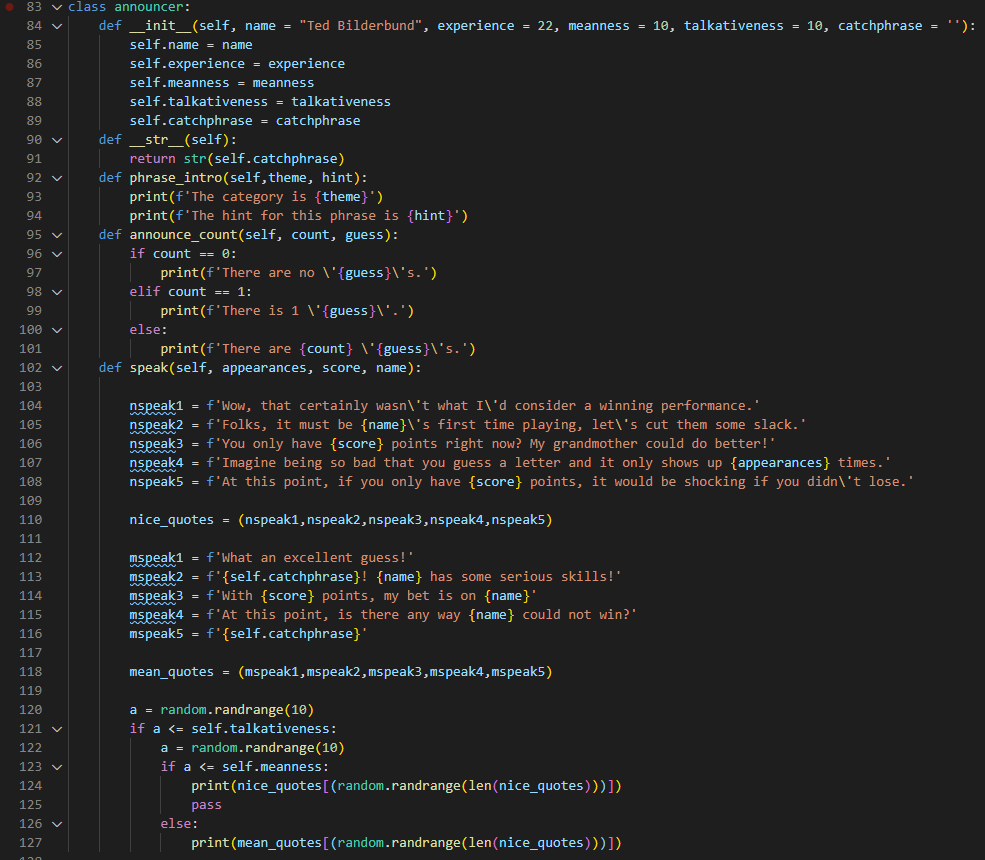
After this, we have print\_guess() which was my bane in two previous projects (3 and 4). It runs by iterating through every element in the guessed set of letters for each single step it takes through the phrase trying to be guessed. This had been where my previous errors were made, as I tried to make everything run in the same step-time which is simply impossible in this manner for this problem.

Lastly, we have appearance\_count which takes the inputted letter and the phrase trying to be guessed, and checks how many times it appears. Initially, I had put the validation step of making sure that the previous letter had not been guessed in this function, but decided that I should leave all input validation at the source of input, rather than waiting until later, so as to minimize confusion to the user.



Our first two classes are those of player and phrase. The player class is rather simple, just keeping data of names, scores, and whether or not they are a real player (the game function takes this into account when determining how to query for a player’s guess).

The second class is a phrase class that has an attribute for the phrase itself, as well as a hint toward the meaning of the phrase, and the theme the phrase was generated in.

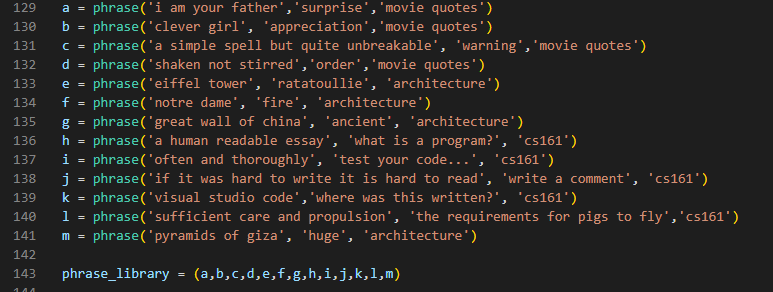


Our last class is a rather large one, largely due to its databank of text. The announcer is one of the most complicated features of this game, with five attributes in total. Three of these attributes deal with front-end interaction (affecting directly what words the announcer says in the game) while the meanness and talkativeness features affect *how* the announcer chooses what to say.

Whenever the announcer has the opportunity to speak in the game (when he isn’t giving tutorializing, one-time information) a randomness function is called in proportion to his talkativeness attribute to determine if he will speak. If he *does* choose to, another randomness function is called that determines whether he will speak positively in proportion to his meanness attribute.

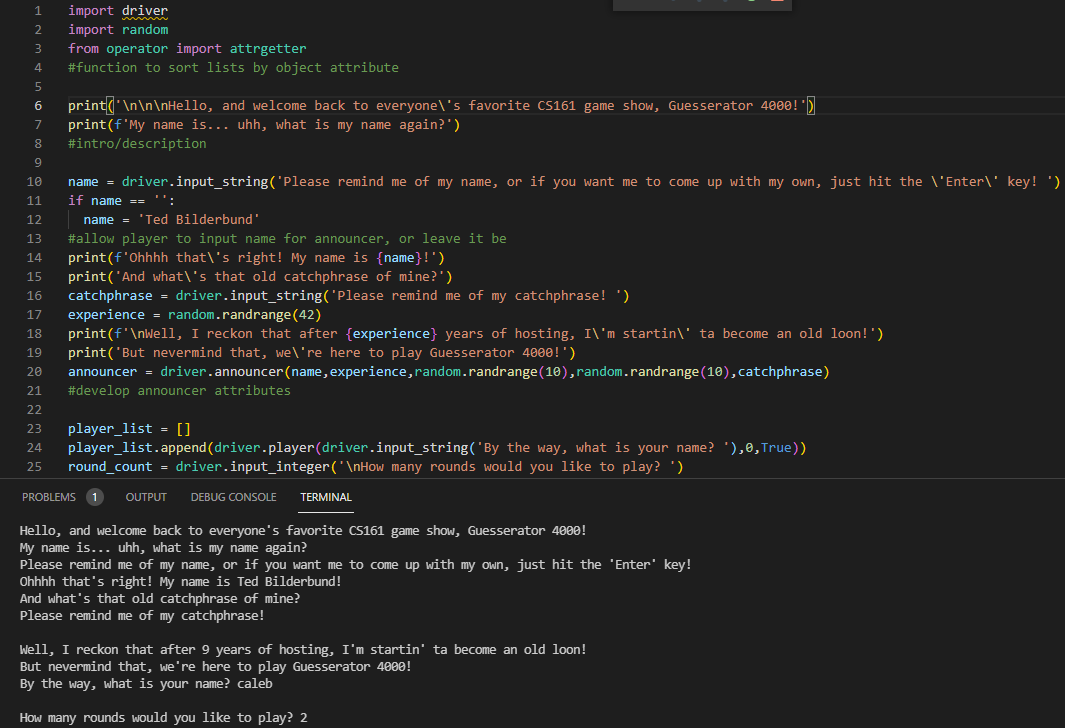
In speaking, the announcer also access real player data and game data to give live feedback, which I think is a pretty cool feature!

Lastly, we have a databank of phrase objects saved as a larger tuple, to be accessed by the main game file.



With this backend out of the way, it is time to look toward…

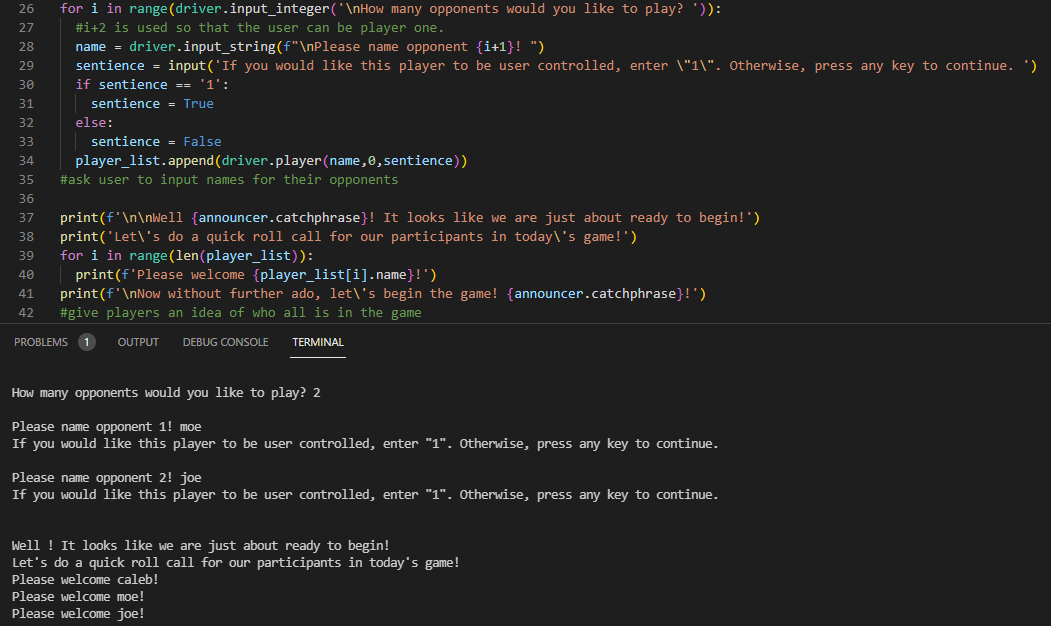
**The Main File:**



In this first screenshot, I have imported my driver file, as well as the random module, and a function called attrgetter. (That’s a surprise tool that will help us later!) We also have some text introducing the game and our announcer, followed by an input for the user to give the announcer a custom name, if they so choose, followed by an input where the user can give the announcer whatever catchphrase they’d like, including nothing at all, that appears later as something I call “enthusiastic silence”.

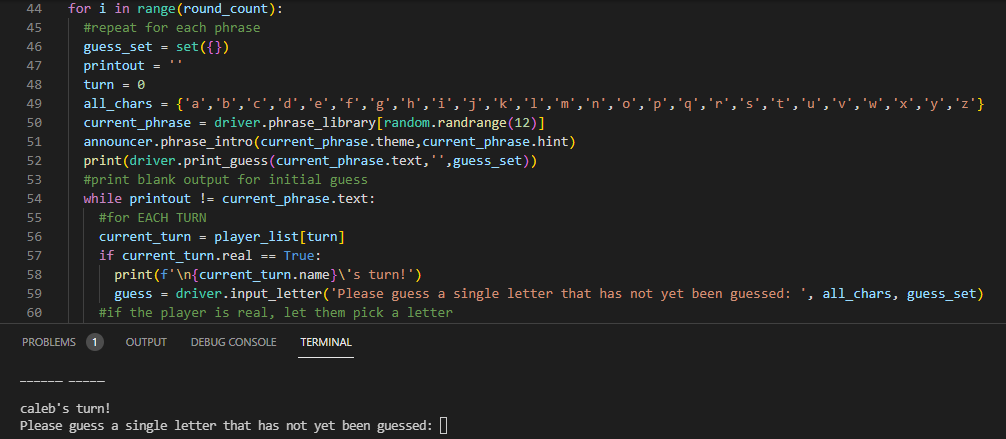
At this point, the announcer randomly generates a number for their years of experience as a host, after which the actual announcer object is created using the previous data and two randrange features placed inside the .announcer class from the driver file.

With the announcer set up, we look to developing an in-game player for our user by asking for them to name themselves, after which they select how many rounds they would like to play.



The user then selects how many opponents they would like to play, which triggers a function that loops for each opposing player. The loop asks for a player’s name and provides the opportunity for the user to add other users as players, to compete against friends, or a mix of friends and CPU players.

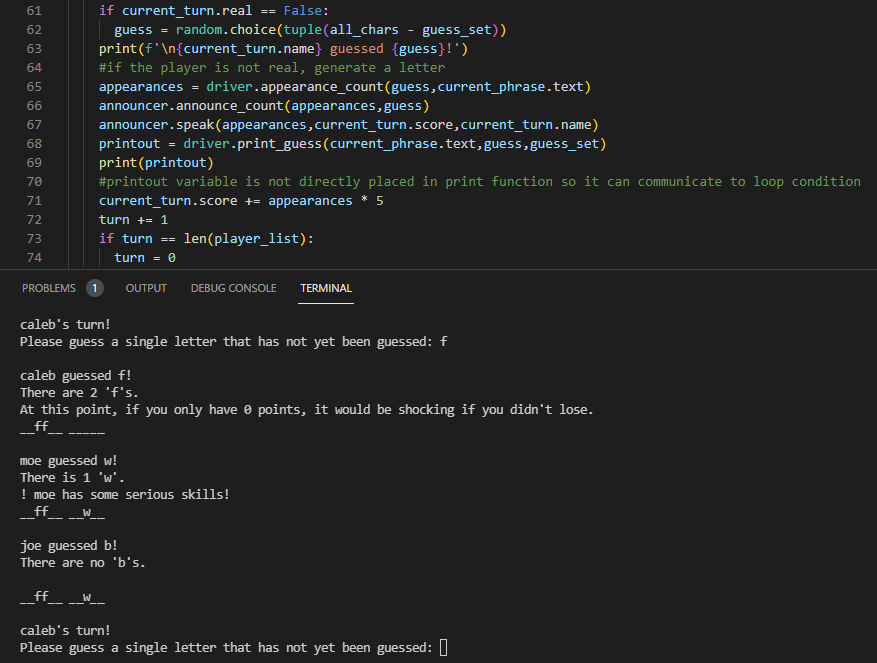
Once all players have been developed and initial conditions are set, the main loop that is the core of the game begins to run.



For the number of rounds specified by our initial user, the program randomly selects a phrase from the tuple-library in the driver file, which is then announced by our announcer.

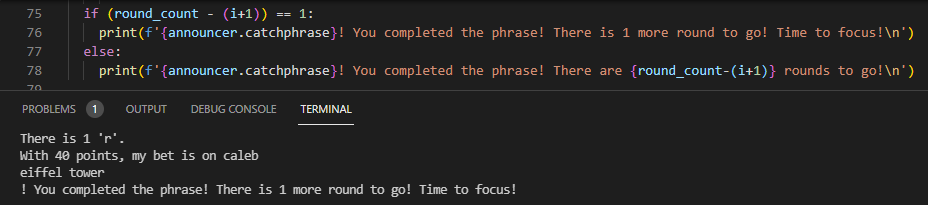
From there, we print out the ‘blocked text’ -> \_\_\_\_\_\_\_ \_\_\_\_ for our players to begin guessing from, and initiate the guessing loop, which runs until the ‘blocked text’ -> b\_o\_\_e\_ \_e\_\_ version equals the phrase itself: ‘blocked text’.

The game cycles through each player, and gives the selected player the opportunity to guess a letter.

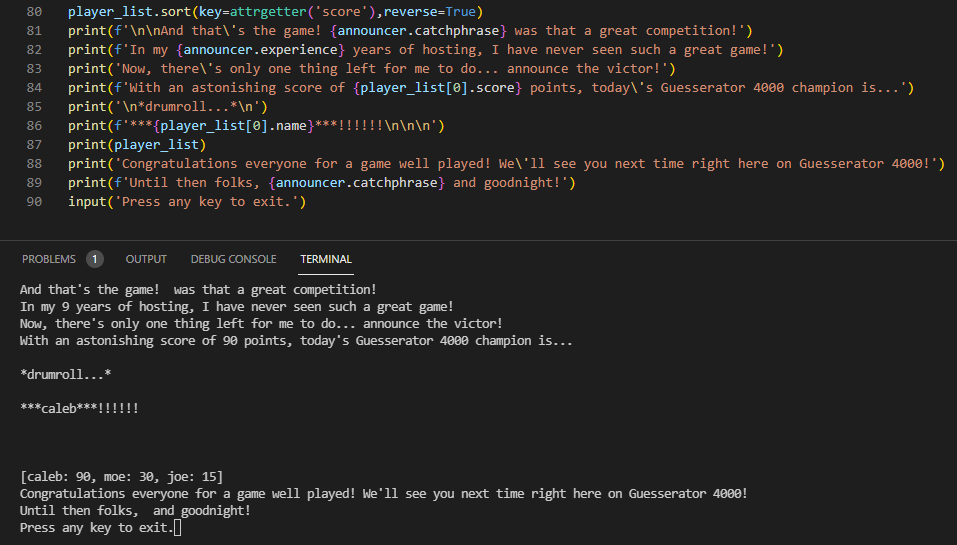


Once a letter is guessed, we print the letter that was guessed, the number of times that letter appears, and provide the opportunity for the announcer to speak, using the .speak() behavior. After this, we print the blocked text that has been updated with the most recent guess, and add the score value of the player’s guess to their score attribute.

Then we iterate to the next player in the cycle and continue the process. (Note that nonreal players use a different guessing process that uses the random.choice function to select a letter that exists in the set of all letters, minus the set of letters that have previously been guessed.)



Once the blocked phrase is identical to the phrase itself, the announcer updates the users on how many rounds they have left, as a sort of text-based progress bar, before beginning the next round and selecting a new phrase again.



Lastly, once all the phrases have been guessed, we use our attrgetter() function to help us sort our players from highest score to lowest. Then we use lots of text as the announcer does announcerey things to build excitement, using attributes from their own object, and eventually announce the winner using attributes of the winning player object via the player list.

The game has been completed, so we leave an unimplemented input to hold the game open until the user decides to close it.

Note to the reader:

Thank you so much for teaching this class. It is incredibly apparent that you love what you do, and are excited to share it with us as students. I had an absolute blast in this class and am more excited about computer science than I ever have been.

Thank you so so much,

~ Caleb Matthews